



Introducing F#

Don Syme
MSR Cambridge



Introducing F#

Don Syme

MSR Cambridge

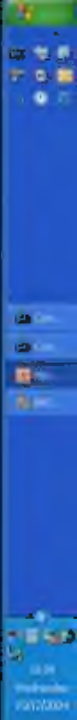
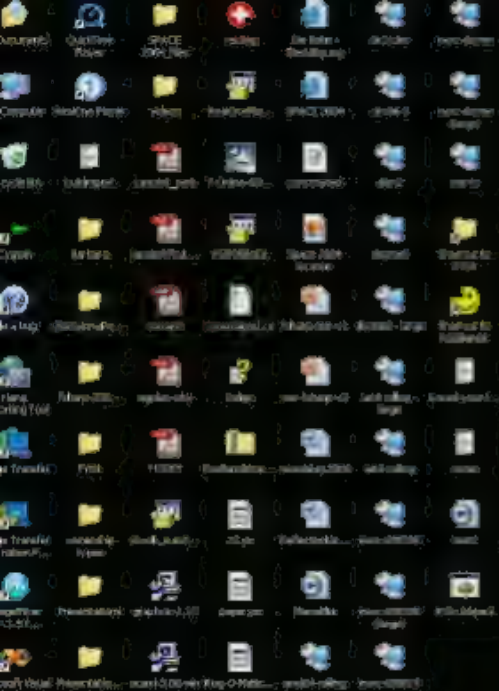
Outline

- F#: a quick orientation via a demo
- F#: orthogonal, simple features
- F#: co-operates in the context of .NET

An ML-F# Orientation

→ DEMO

- 1. Write a symbolic differentiator
- 2. Write an evaluation function
- 3. Make it extensible with new operators
- 4. Display sample plots on a form
- 5. Use the differentiator from C#



Visual Studio IDE interface showing the Object Browser, Solution Explorer, and Output windows.

Object Browser

Shows the hierarchy of the current project. The left pane lists the project components, and the right pane displays the details of the selected component.

Solution Explorer

Shows the structure of the solution. The left pane lists the solution components, and the right pane displays the details of the selected component.

Output

Shows the output of the application. The left pane lists the output categories, and the right pane displays the output text.

Toolbox

Shows the toolbox for the application. The left pane lists the toolbox categories, and the right pane displays the toolbox controls.

Properties Window

Shows the properties of the selected component. The left pane lists the property categories, and the right pane displays the property values.

Code Window

Shows the source code of the application. The left pane lists the code files, and the right pane displays the code text.

Command Window

Shows the command line interface. The left pane lists the command categories, and the right pane displays the command text.

Error List

Shows the list of errors in the application. The left pane lists the error categories, and the right pane displays the error details.

Output

Shows the output of the application. The left pane lists the output categories, and the right pane displays the output text.

```
(* Sample F# source file *)
```

```
open System
```

```
type expr =
```

```
| Sum of expr * expr  
| Prod of expr * expr  
| Const of system.Double  
| Var of string
```

```
let eval env expr = I  
    match expr with  
    | Const d
```

File Edit View Project Build Debug Tools Window Help

(* Sample F# Source File *)

open System

type expr =

- | Sum of expr * expr
- | Prod of expr * expr
- | Const of system.Double
- | Var of string

let eval env expr =

match expr with| Const c -> c

Output

Download from

F#CC - Developer edition, all third-party packages allowed to load.

File1.cs - Class Browser

(* Sample F# Source File *)

open System

type expr =

- | Sum of expr * expr
- | Prod of expr * expr
- | Const of system.Double
- | Var of string

let eval env expr =

match expr with

- | Const c -> c
- | Sum(e1,e2) -> eval env I

Output

Show output from

F#CC - Developer edition, all third-party packages allowed to load.


```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}

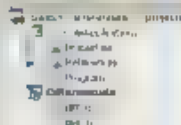
```

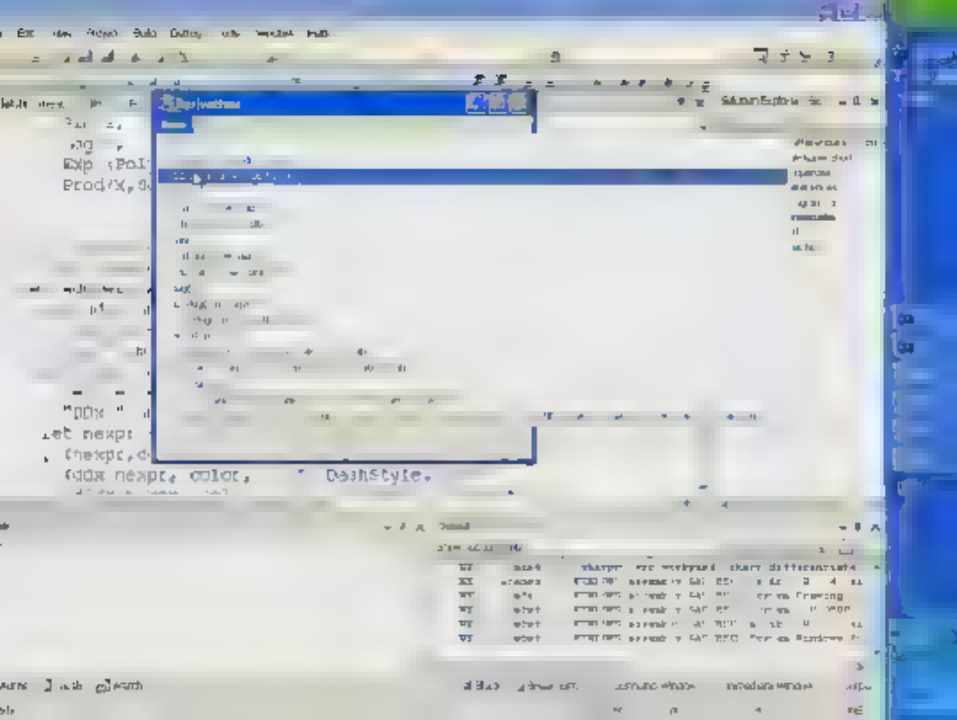


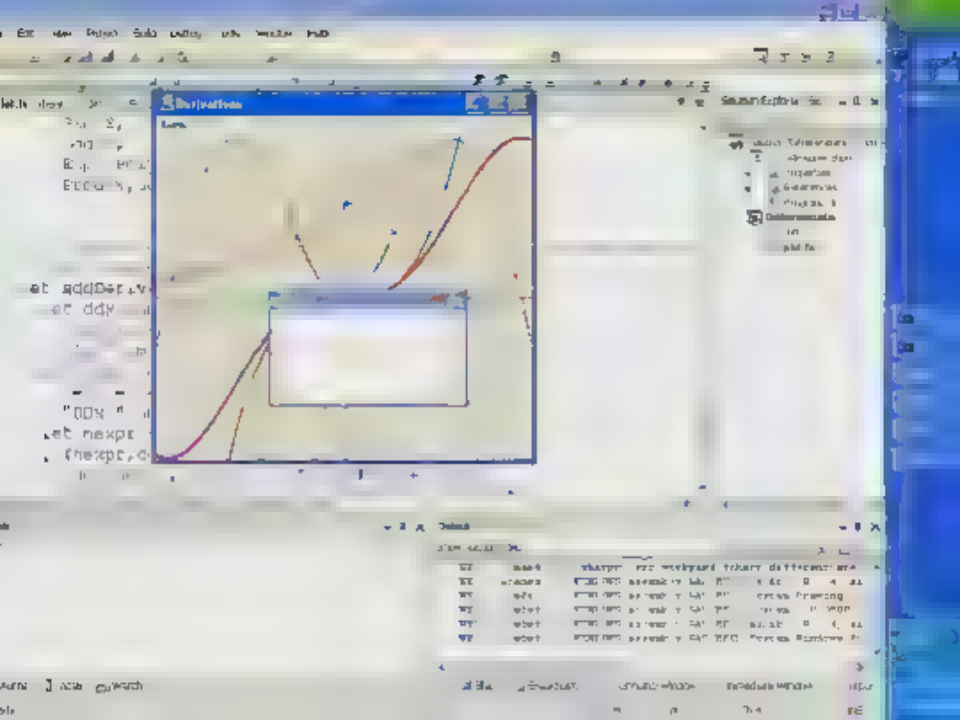
```
type Expr =
  Sum of Expr * Expr
  Prod of Expr * Expr
  Var of string
  ...
```

```
and JExprParser =
```

```
  Evaluate : Expr -> float
  Differentiate : Expr -> Expr
  Print : Expr -> string
  Name : string
```







2. 9. 2001/2002 9. 10. 2001/2002

2000 10 10

DIFF - MYG.DIFFEV.DIFFERENT.gto.

100 Expt = Myocardial Differentiate Expt

የጥቅም

[illegible]

4

Program

Expr	-	Diff	/	Diff	-	Diff	Expr
------	---	------	---	------	---	------	------

Command \rightarrow Diff

Case No.	Case Name	Case Type	Case Status
1	Case 1	Case 1	Case 1
2	Case 2	Case 2	Case 2
3	Case 3	Case 3	Case 3
4	Case 4	Case 4	Case 4
5	Case 5	Case 5	Case 5
6	Case 6	Case 6	Case 6
7	Case 7	Case 7	Case 7
8	Case 8	Case 8	Case 8
9	Case 9	Case 9	Case 9
10	Case 10	Case 10	Case 10
11	Case 11	Case 11	Case 11
12	Case 12	Case 12	Case 12
13	Case 13	Case 13	Case 13
14	Case 14	Case 14	Case 14
15	Case 15	Case 15	Case 15
16	Case 16	Case 16	Case 16
17	Case 17	Case 17	Case 17
18	Case 18	Case 18	Case 18
19	Case 19	Case 19	Case 19
20	Case 20	Case 20	Case 20
21	Case 21	Case 21	Case 21
22	Case 22	Case 22	Case 22
23	Case 23	Case 23	Case 23
24	Case 24	Case 24	Case 24
25	Case 25	Case 25	Case 25
26	Case 26	Case 26	Case 26
27	Case 27	Case 27	Case 27
28	Case 28	Case 28	Case 28
29	Case 29	Case 29	Case 29
30	Case 30	Case 30	Case 30
31	Case 31	Case 31	Case 31
32	Case 32	Case 32	Case 32
33	Case 33	Case 33	Case 33
34	Case 34	Case 34	Case 34
35	Case 35	Case 35	Case 35
36	Case 36	Case 36	Case 36
37	Case 37	Case 37	Case 37
38	Case 38	Case 38	Case 38
39	Case 39	Case 39	Case 39
40	Case 40	Case 40	Case 40
41	Case 41	Case 41	Case 41
42	Case 42	Case 42	Case 42
43	Case 43	Case 43	Case 43
44	Case 44	Case 44	Case 44
45	Case 45	Case 45	Case 45
46	Case 46	Case 46	Case 46
47	Case 47	Case 47	Case 47
48	Case 48	Case 48	Case 48
49	Case 49	Case 49	Case 49
50	Case 50	Case 50	Case 50
51	Case 51	Case 51	Case 51
52	Case 52	Case 52	Case 52
53	Case 53	Case 53	Case 53
54	Case 54	Case 54	Case 54
55	Case 55	Case 55	Case 55
56	Case 56	Case 56	Case 56
57	Case 57	Case 57	Case 57
58	Case 58	Case 58	Case 58
59	Case 59	Case 59	Case 59
60	Case 60	Case 60	Case 60
61	Case 61	Case 61	Case 61
62	Case 62	Case 62	Case 62
63	Case 63	Case 63	Case 63
64	Case 64	Case 64	Case 64
65	Case 65	Case 65	Case 65
66	Case 66	Case 66	Case 66
67	Case 67	Case 67	Case 67
68	Case 68	Case 68	Case 68
69	Case 69	Case 69	Case 69
70	Case 70	Case 70	Case 70
71	Case 71	Case 71	Case 71
72	Case 72	Case 72	Case 72
73	Case 73	Case 73	Case 73
74	Case 74	Case 74	Case 74
75	Case 75	Case 75	Case 75
76	Case 76	Case 76	Case 76
77	Case 77	Case 77	Case 77
78	Case 78	Case 78	Case 78
79	Case 79	Case 79	Case 79
80	Case 80	Case 80	Case 80
81	Case 81	Case 81	Case 81
82	Case 82	Case 82	Case 82
83	Case 83	Case 83	Case 83
84	Case 84	Case 84	Case 84
85	Case 85	Case 85	Case 85
86	Case 86	Case 86	Case 86
87	Case 87	Case 87	Case 87
88	Case 88	Case 88	Case 88
89	Case 89	Case 89	Case 89
90	Case 90	Case 90	Case 90
91	Case 91	Case 91	

9. $\frac{1}{2} \pi$

[illegible][illegible]

The program 105 disseminates and 107 has entered with some

1

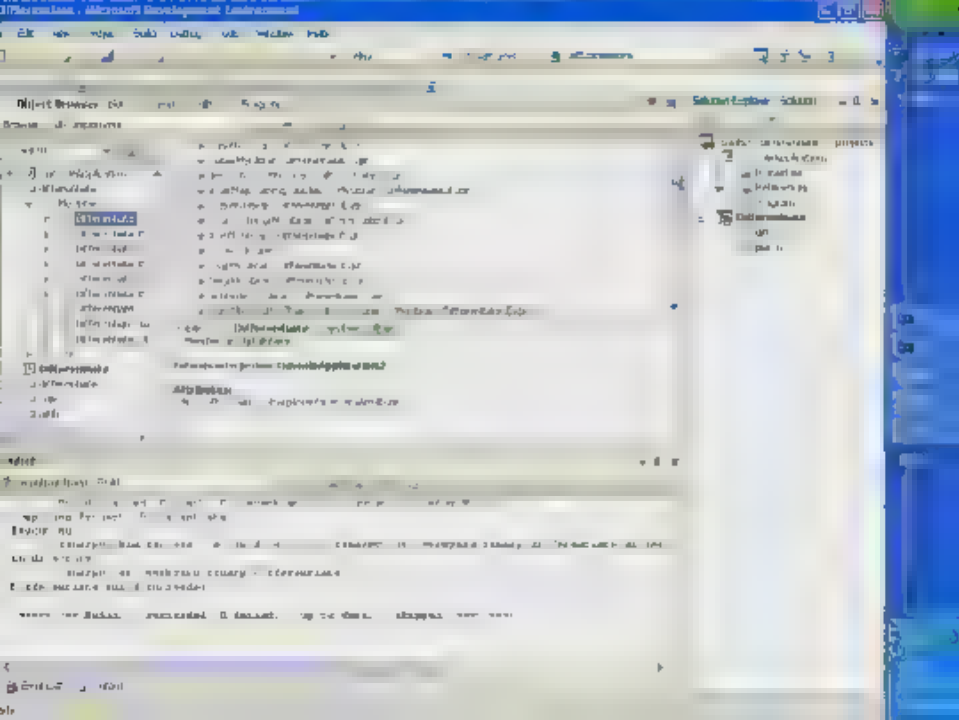
501.

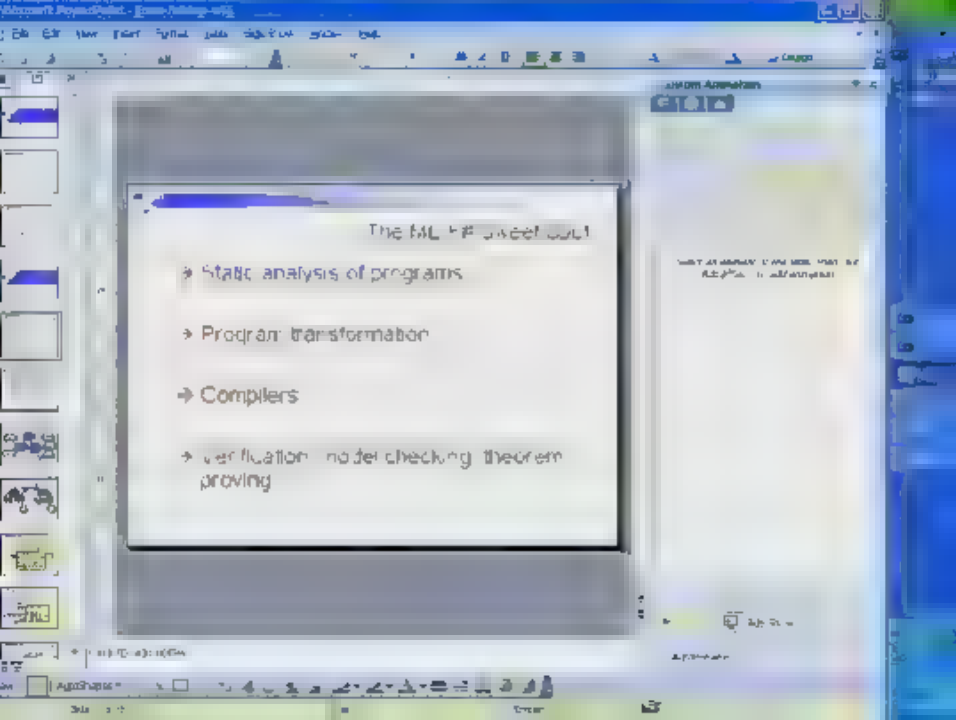
2

418

74

re





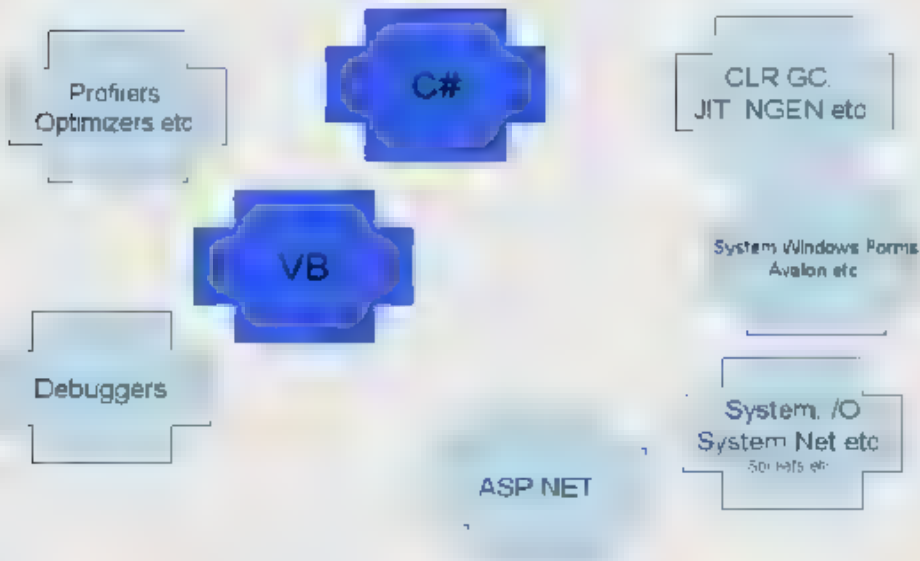
The ML # Sweet Spot

- Static analysis of programs
- Program transformation
- Compilers
- Verification (model checking, theorem proving)

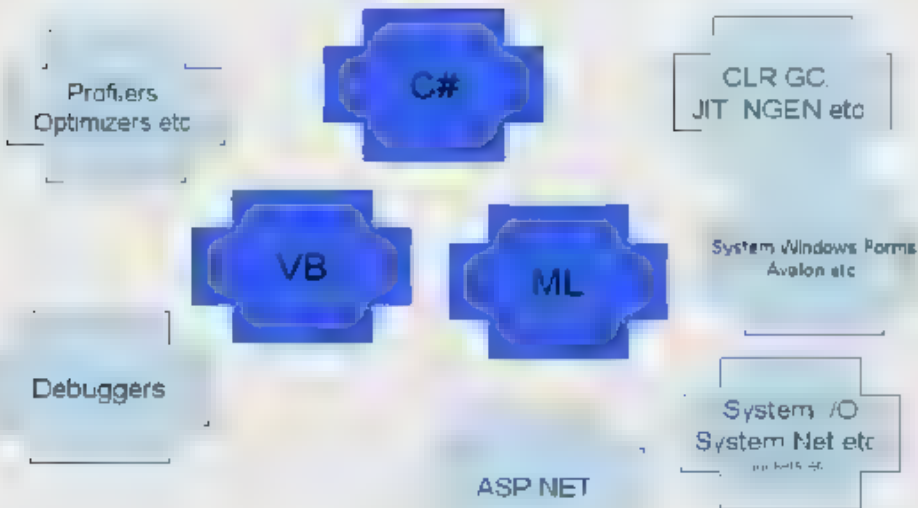
ML in Microsoft

- Static Driver Verifier (Core OS)
- High-level model of FRS (Core File Services)
- Zap theorem prover (MSR)
- KIS race condition finder (MSR)

F# within .NET



F# within .NET



F# as a Language

OCaml

F#

F# as a Language



OCaml



F#

F# as a Language



OCaml



F#

F# as a Language



• `embObjects`
• `libObj` / `poly`
• `used extensions`

Modules-as-
values

`let module Foo = ... and
let module Bar = ...`

OCaml



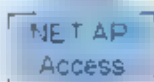
F#

F# as a Language



F# as a Language

Powerful, simple programming language



OCaml

+ tools

F#

+ tools

Orthogonal & Unified Constructs

→ Tuples = multiple arguments = multiple returns

Tuples
in C++

```
et args = (1,2,3)
```

```
et f (a,b,c) = (a+b , b+c, a+b)
```

```
et x,y,z = f(f(f args))
```

```
do printf "res = %d,%d,%d" x y z  
res = 17 16 15"
```

Orthogonal & Unified Constructs

→ Tuples = multiple arguments = multiple returns

`e+ args = (1,2,3)`

Tuples
as Data

Multiple-Args

`e+ f (a,b,c) = (a+b , b+c, a+b)`

Multiple
Returns

`e+ x,y,z = f(f(f args))`

`do printf "res = %d,%d,%d" x y z
res = 17 16 15`

Orthogonal & Unified Constructs

→ Let “let” simplify your life...

```
let data = (1,2,3)
```

```
let f(a,b,c) =
```

```
  let sum = a + b + c in
```

```
  let g(x) = sum + x*x in
```

```
  g(a), g(b), g(c)
```

Orthogonal & Unified Constructs

→ Let "let" simplify your life.

Bind a state to a value

```
let data = (1,2,3)
```

```
let f(a,b,c) =
```

```
  let sum = a + b + c in
```

```
  let g(x) = sum + x*x in  
  g(a), g(b), g(c)
```


Orthogonal & Unified Constructs

→ Let "let" simplify your life...

Bind a static value

```
let data = (1,2,3)
```

Bind a static function

```
let f(a,b,c) =
```

```
  let sum = a + b + c in
```

```
  et g(x) = sum + x*x in
```

```
  g(a), g(b), g(c)
```

Bind a local value

Bind a local function

Orthogonal & Unified Constructs

- “let” + “capture”: sophisticated operations in sophisticated contexts

```
let readBinary(inputStream) =  
  let read () = inputStream.ReadByte in  
  let smallFormat = (read() == 0x2) in  
  
  let readOneRecord() =  
  
    if (smallFormat) then read()  
    else ... in  
  
  readOneRecord().  
  readOneRecord().
```

Orthogonal & Unified Constructs

- “let” + “capture”: sophisticated operations in sophisticated contexts.

Context

Operation in
Context?

```
let readBinary(inputStream) =  
  let read () = inputStream.ReadByte in  
  let smallFormat = (read() == 0x2) in
```

```
    let readOneRecord() =
```

```
      if (smallFormat) then read()  
      else ... in
```

```
readOneRecord():  
readOneRecord().
```

Orthogonal & Unified Constructs

- “let” + “capture”, sophisticated operations in sophisticated contexts .

```
let readBinary(inputStream) =  
  let read () = inputStream.ReadByte() in  
  let smallFormat = (read() == 0x2) in
```

```
    let readOneRecord() =  
      if (smallFormat) then read()  
      else ... in
```

```
  readOneRecord().  
  readOneRecord().
```

Context

Operation in
Context

Operation in
Sub-Context

Orthogonal & Unified Constructs

- “let” + “capture”: sophisticated operations in sophisticated contexts. .

```
let readBinary(inputStream) =  
  let read () = inputStream.ReadByte() in  
  let smallFormat = (read() == 0x2) in
```

```
    let readOneRecord() =  
      if (smallFormat) then read()  
      else ... in
```

```
  readOneRecord().  
  readOneRecord().
```

Context

Operation in
Context

Operation in
Other context

Let's generalize
this context

Orthogonal & Unified Constructs

→ Immutability the norm..

Orthogonal & Unified Constructs

→ Immutability the norm..

```
let data = 3  
do data = 4
```

Orthogonal & Unified Constructs

→ Immutability the norm.

```
let data = 3
```

```
do data = 4
```

```
type data =
```

```
{ Name: string;
```

```
  Size: int;
```

```
  Items: Map<string,string> }
```

```
let NewData(name,items) =
```

```
{ Name=name
```

```
  Size=Map size items,
```

```
  Items=items }
```


Orthogonal & Unified Constructs

→ Immutability the norm...

All F# data is
immutable by
default

```
let data = 3  
do data = 4
```

Simple, robust, and
not too changed

```
type data =  
{ Name: string;  
  Size: int;  
  Items: Map<string,string> }
```

```
let NewData(name, items) =  
{ Name=name;  
  Size=Map.size items;  
  Items=items }
```

In praise of immutability

- Immutable objects can be relied on
- Immutable objects can transfer between threads
- Immutable objects can be aliased safely
- Immutable objects can be optimized

Orthogonal & Unified Constructs

- F# is a functional language
- F# is an imperative language
- **F# is a mixed functional/imperative language**
- F# is not Haskell. Loops encouraged

Orthogonal & Unified Constructs

- **Function** values simplify and unify
 . . iteration

Orthogonal & Unified Constructs

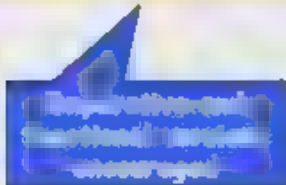
→ **Function** values simplify and unify
.iteration

```
val List map: ('a -> 'b) -> list<'a> -> list<'b>  
val List foreach: list<'a> -> ('a -> unit) -> unit  
val IEnumerable map: ('a -> 'b) -> IEnumerable<'a> -> IEnumerable<'b>  
val IEnumerable foreach: IEnumerable<'a> -> ('a -> unit) -> unit
```

Orthogonal & Unified Constructs

→ Function values simplify and unify
.iteration

```
val List map. ('a -> 'b) -> list<'a> -> list<'b>  
val List foreach. list<'a> -> ('a -> unit) -> unit  
val IEnumerable map: ('a -> 'b) -> IEnumerable<'a> -> IEnumerable<'b>  
val IEnumerable foreach. IEnumerable<'a> -> ('a -> unit) -> unit
```



Orthogonal & Unified Constructs

→ Function values simplify and unify
.iteration

```
val List map: ('a -> 'b) -> list<'a> -> list<'b>  
val List foreach: list<'a> -> ('a -> unit) -> unit  
val IEnumerable map: ('a -> 'b) -> IEnumerable<'a> -> IEnumerable<'b>  
val IEnumerable foreach: IEnumerable<'a> -> ('a -> unit) -> unit
```



Orthogonal & Unified Constructs

- Function values simplify and unify
 - . .extension

```
type UnaryOperator =  
  ( evaluate. (float → float),      // evaluation function  
    differentiate. (expr → expr), // symbolic differentiation function  
    name. string )                 // identity  
  
let sinop = { evaluate=sin,  
              differentiate=Cos,  
              name="sin" }
```


Orthogonal & Unified Constructs

- Function values simplify and unify
 - . .extension

```
type UnaryOperator =  
  ( evaluate. (float → float),      // evaluation function  
    differentiate. (expr → expr), // symbolic differentiation function  
    name. string )                  // identity  
  
let sinop = { evaluate=sin,  
              differentiate=Cos,  
              name="sin" }
```

Orthogonal & Unified Constructs

→ Function values simplify and unify
.extension

```
type UnaryOperator =  
  ( evaluate. (float → float),      // evaluation function  
    differentiate. (expr → expr), // symbolic differentiation function  
    name. string )                  // identity  
  
let sinop = { evaluate=sin,  
              differentiate=Cos,  
              name="sin" }
```

Orthogonal & Unified Constructs

- Function values simplify and unify
 - . .extension

```
type UnaryOperator =  
  ( evaluate. (float → float),      // evaluation function  
    differentiate. (expr → expr),    // symbolic differentiation function  
    name. string )                  // identity  
  
let sinop = { evaluate=sin,  
              differentiate=Cos,  
              name="sin" }
```

subclass?

Orthogonal & Unified Constructs

- Function values simplify and unify
 . .extension

```
type UnaryOperator =  
  ( evaluate. (float → float),      // evaluation function  
    differentiate. (expr → expr), // symbolic differentiation function  
    name. string )                 // identity  
  
let sinop = { evaluate=sin,  
              differentiate=Cos;  
              name="sin" }
```

Method methods
Confused? Remember that
you have a pointer to
the object

Create Subclass
with
hard

subclass?

Orthogonal & Unified Constructs

→ Type parameters

```
Map<'a,'b>  
List<'a>  
Set<'a>
```

→ Discriminated unions

```
type expr =  
  | Sum of expr * expr  
  | Prod of expr * expr
```

→ Pattern matching

→ Type inference

→ Recursion (Mutually-referential objects)

Orthogonal & Unified Constructs

→ Type parameters

```
Map<'a,'b>  
List<'a>  
Set<'a>
```

→ Discriminated unions

```
type expr =  
  | Sum of expr * expr  
  | Prod of expr * expr
```

→ Pattern matching

→ Type inference

```
match expr with  
  | Sum(a,b) ->  
  | Prod(a,b) ->
```

→ Recursion (Mutually-referential objects)

```
let rec map =
```

Less is More?

- Fewer concepts = Less time in class design
- No null pointers¹
- Far fewer classes and other type definitions
- No constructors-calling-virtual-methods and other OO dangers

1. No exception handling across from .NET APIs

Problems are problems

- Unmanaged resources (IDisposable)
- Exceptions
- I/O, avoiding blocking, concurrency
- API design
- Complex software is, well, complex...

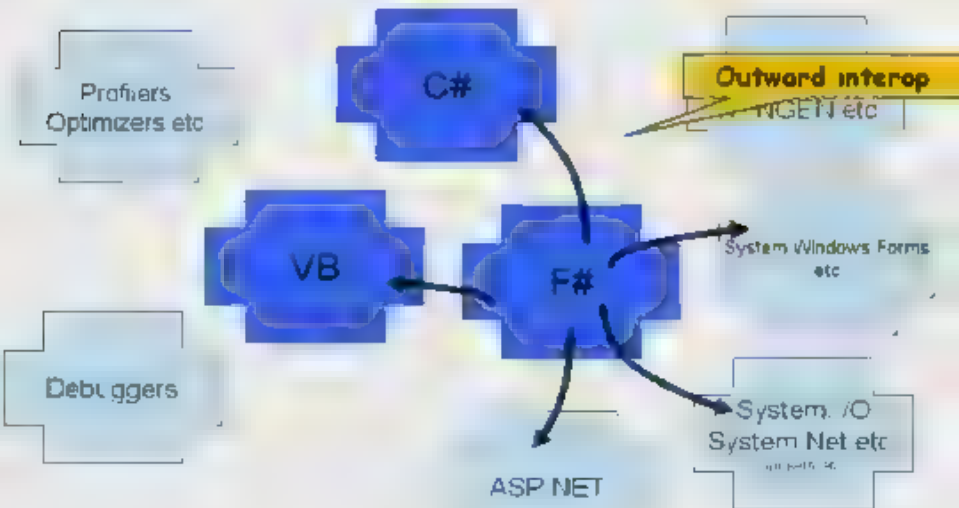
F# Observations

- F# gives you the tools to concentrate on the algorithmic complexities of symbolic processing
- ➔ F# is:
 - excellent for using the λ calculus
 - excellent for writing ML libraries
 - not using ML libraries: safe from λ = T
- So the niche seems to be for writing sophisticated applications
 - probably, with some use of the components
 - probably, with some symbolic processing
 - probably with some components written in #

Part 2 F# Co-operates

The F# challenge: make it fit with .NET in a deeply practical way without losing the essential goodness of ML programming

Connecting F# to .NET



Outward Interop

→ A very nice "." notation

```
System.Console.WriteLine("abc")  
form.SetStyle(ControlStyles.AllPaintingInWmPaint, true);  
form.ClientSize = new System.Drawing.Size(292, 266)
```

→ Quite similar to C#

→ Delegates created using function values

Outward Interop

- Object expressions \Rightarrow closures for objects

```
let myForm title n =  
  ( new System.Windows.Forms.Form() as base  
    with OnPaint(args) =  
      base.OnPaint(args).  
      Console.WriteLine("OnPaint\n")  
    and OnResize(args) =  
      base.OnResize(args).  
      Console.WriteLine("OnResize args = {0}\n" args)  
  )
```

- Overriding only
- Orthogonal type inference, capture mutually recursive nested

1. Introduce the topic and state the purpose of the presentation.

2. Introduce the topic and state the purpose of the presentation.

3. Introduce the topic and state the purpose of the presentation.

4. Introduce the topic and state the purpose of the presentation.

5. Introduce the topic and state the purpose of the presentation.

6. Introduce the topic and state the purpose of the presentation.

7. Introduce the topic and state the purpose of the presentation.

8. Introduce the topic and state the purpose of the presentation.

9. Introduce the topic and state the purpose of the presentation.

10. Introduce the topic and state the purpose of the presentation.

Outward Interop

- Object expressions → closures for objects

```
let myForm title n =  
  ( new System.Windows.Forms.Form() as base  
    with OnPaint(args) =  
      base.OnPaint(args).  
      Console.WriteLine ("OnPaint\n")  
    and OnResize(args) =  
      base.OnResize(args).  
      Console.WriteLine ("OnResize args = {0}\n" args)  
  )
```

- Overriding only
- Orthogonal type inference, capture mutually recursive nested

Inward Interop

The screenshot shows a Java IDE window titled 'person.java' with a class hierarchy for 'Person'. The class is annotated with '@InwardInterop' and '@InwardInteropClass'. The class has several methods, including 'getPersonId', 'setPersonId', 'getPersonName', 'setPersonName', 'getPersonAge', 'setPersonAge', 'getPersonSex', 'setPersonSex', 'getPersonAddress', 'setPersonAddress', and 'getPersonPhone'. The class is also annotated with '@InwardInteropClass'.

Annotations shown in the code:

- `@InwardInterop`
- `@InwardInteropClass`

Methods shown in the code:

- `getPersonId()`
- `setPersonId(int personId)`
- `getPersonName()`
- `setPersonName(String personName)`
- `getPersonAge()`
- `setPersonAge(int personAge)`
- `getPersonSex()`
- `setPersonSex(String personSex)`
- `getPersonAddress()`
- `setPersonAddress(String personAddress)`
- `getPersonPhone()`
- `setPersonPhone(String personPhone)`

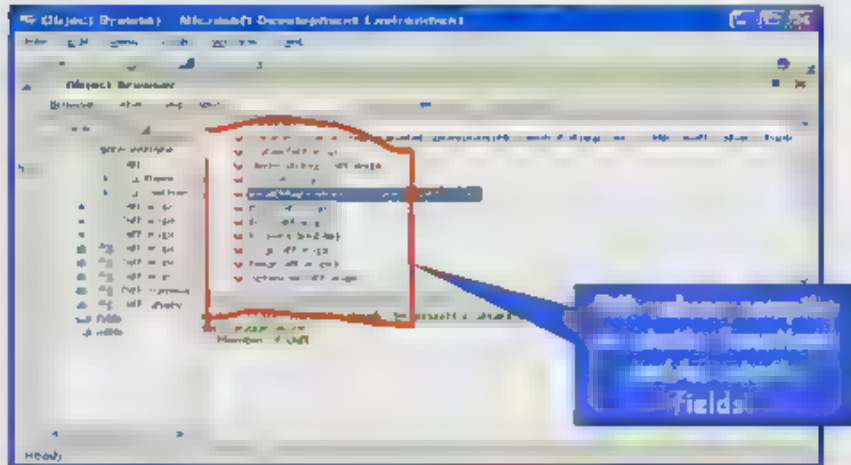
Annotations shown in the comments:

- `@InwardInteropClass`

Inward Interop
Inward Interop Class
Inward Interop Class

Inward Interop
Inward Interop Class
Inward Interop Class
Inward Interop Class

Inward Interop



Inward Interop

- Inward Interop is **essential** for many reasons, e.g. **componentization, testing, profiling**
- No other ML implementation in the world has it
- Inward interop gives you full access, but is sometimes a little awkward from C#
- Room for extensions to what is there

What else does F# offer? ☺

→ Libraries galore

- GL's Network Speech Graphics

→ Tools galore

- CPU Profiling Memory Profiling Debugging
- Visual Studio integration

→ C# next door

- Fantastic interop with C and COM

→ Components

- Can build versionable binary compatible DLLs

→ Multi-threading that works

- Even OCaml has problems here

→ No significant runtime components

- GC etc is not part of the package

What else does F# offer? ☺

→ Libraries galore

GL's Network Speech Graphics



→ Tools galore

CPU Profiling Memory Profiling Debugging
Visual Studio integration

→ C# next door

Fantastic interop with C and COM

→ Components

Can build versionable binary compatible DLLs

→ Multi-threading that works

Even OCaml has problems here

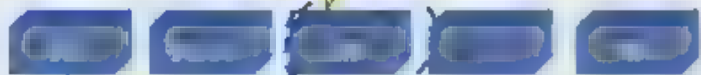
→ No significant runtime components

GC etc is not part of the package

F#, the CLR & .NET Generics



F#, t and here R & .NET Generics



also here

We've added support for
generics/polymorphism...

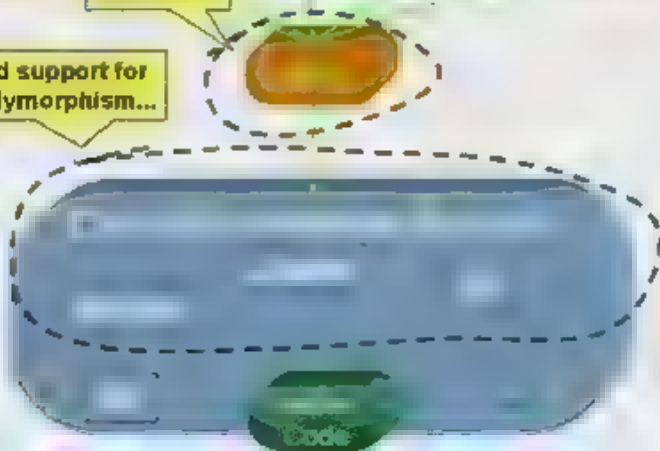


F#, t and here R & .NET Generics

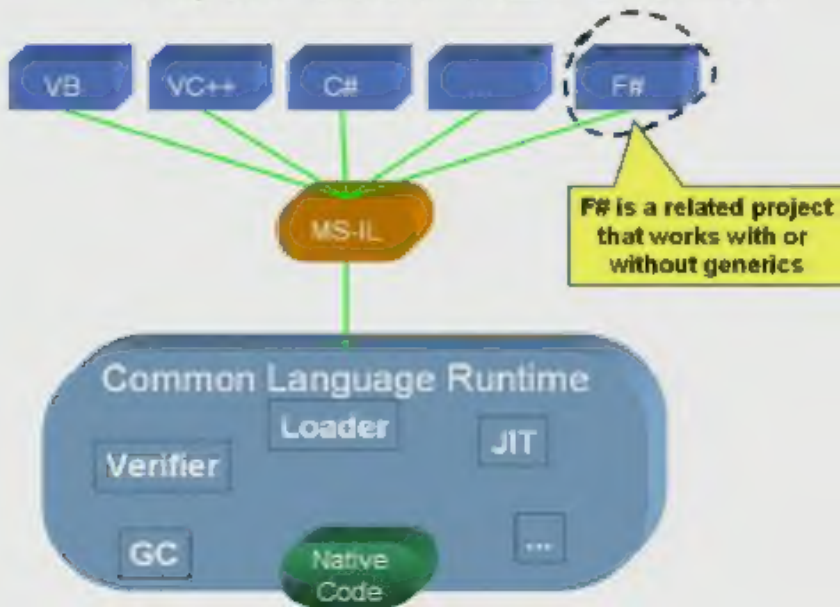


also here

We've added support for
generics/polymorphism...



F#, the CLR & .NET Generics



F# v1.0

- Very stable core language and compiler
 - ▣ Some interesting language work will go on around the edges, e.g. ML-modules-without-ML-modules
- Being used by Zapato & SDV & many others
- ~6000 downloads/year
- VisualStudio 2005 Beta 1 integration
- ML compatibility library
- Samples etc.
- Tools: Lexer, Parser Generators

F# & Research

- Practical programming in ML still raises many interesting language design questions
- e.g. mutually recursive initialization that compensate correctly when failure occurs half-way through
- F# is a great place to conduct this research
 - easy to code examples of real-world practical programming
 - a simple clean language to extend

Questions?

<http://research.microsoft.com/projects/fsharp>

<http://research.microsoft.com/projects/fsharp>

autogroup: <mailto:fsharp>

\\cam-01-srv\dfsroot\projects\fcom\releases